CEWES MSRC/PET TR/98-05

Using the MPE Graphics Library with Fortran90

by

S. W. Bova
Clay P. Breshears

DoD HPC Modernization Program
Programming Environment and Training
CEWES MSRC

CEWES MSRC

Nichols Research

02h00298

**Work funded by the DoD High Performance Computing
Modernization Program CEWES
Major Shared Resource Center through**

Programming Environment and Training (PET)

Supported by Contract Number: DAHC 94-96-C0002
Nichols Research Corporation

Views, opinions, and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of Defense Position, policy, or decision unless so designated by other official documentation.

Using the MPE Graphics Library with Fortran90

S.W. Bova[*] Clay P. Breshears[†]

March 13, 1998

1 Introduction

The MPE graphics library is part of the MPICH package distributed by Argonne National Laboratory. "It consists of functions that are not in MPI; consistent in style with MPI; freely available; and in the long run will work with any MPI implementation" [1]. This graphics library gives the MPI programmer an easy-to-use, minimal set of routines that can asynchronously draw color graphics to an X11 window during the course of a numerical simulation. Unfortunately, there is little scientific visualization software written for MPE, which places a high burden on the applications programmer. This difficulty is compounded by a dearth of documentation on the library. On the other hand, the library is small and has a shallow learning curve. This report is a brief description of how these graphics routines may be called from a Fortran90 program, written from the perspective of a finite element applications programmer. It was written after gaining one week of experience in the use of the MPE graphics library, thereby demonstrating that the library is compact enough to be easily used if the applications programmer is familiar with graphics programming. A goal of this report is to make the package equally useful to those programmers who are not as familiar with graphics programming. Note it is not the intent of this report to provide a complete programmer's guide to the MPE library.

The discussion begins in Section 2 with a description of some of the library functions. Next, Section 3 describes a Fortran90 module which uses these routines to draw colored contour plots for two-dimensional, unstructured grids. Section 4 presents an overview on methods and software which can be used to save the graphics for later viewing. Finally, a summary of this report is presented in Section 5 with instructions on how to obtain the module and the MPE graphics library.

2 MPE Graphics Library

In this section, a brief discussion of a subset of the MPE graphics library is presented. For the details of the bindings, refer to the Unix `man` pages which are included in the MPE distribution. Portability issues are also discussed.

[*]CEWES MSRC On-site CFD Lead for PET, Mississippi State University
[†]CEWES MSRC On-site Parallel Tools Lead for PET, Rice University

`MPE_Open_graphics()`	`MPE_Draw_circle()`
`MPE_CaptureFile()`	`MPE_Add_RGB_color()`
`MPE_Close_graphics()`	`MPE_Draw_point()`
`MPE_Update()`	`MPE_Draw_points()`
`MPE_Draw_line()`	`MPE_Draw_logic()`
`MPE_Make_color_array()`	`MPE_Fill_circle()`
`MPE_Line_thickness()`	`MPE_Fill_rectangle()`
`MPE_Num_colors()`	

Table 1: MPE graphics routines that have Fortran bindings.

2.1 MPE Graphics Routines

The routines given in Table 1 have Fortran bindings and are available in the current release of MPE graphics. The first six of these routines in the left-hand column were used in the Fortran90 module described in Section 3. The purpose of most of the routines should be obvious from their names. The routine **`MPE_Open_graphics()`** should always be called first. It initializes the system and opens a window on a specified X11 display. The last routine called should be **`MPE_Close_graphics()`**, which gracefully shuts down the X11 window. A color map may be easily defined with **`MPE_Make_color_array()`**. This routine returns color indices in a predefined spectrum. (Arbitrary colors may be defined using **`MPE_Add_RGB_color()`**, but this is not necessary if the colors returned by **`MPE_Make_color_array()`** are satisfactory.) Lines between two points are drawn with **`MPE_Draw_line()`**. The purpose of **`MPE_Update()`** is perhaps not as obvious. Graphics requests are buffered in order to improve performance. A call to **`MPE_Update()`** flushes the graphics buffer and ensures that objects which are drawn are actually displayed in the X11 window. Finally, it may be desirable to capture the contents of the X11 window in an image file on disk. This action is toggled by calling **`MPE_Capturefile()`** which uses the X11 utility **`xwd`**. One of the required arguments is a frequency parameter; *e.g.* if this parameter is equal to two, then an image will be captured every other time **`MPE_Update()`** is called.

The following two routines, along with their Fortran bindings, were added for the present work and are not contained in the current MPE release: **`MPE_Fill_triangle()`** and **`MPE_Fill_polygon()`**. Since these routines are not part of the MPE library, their Fortran and C bindings are given in Figures 1 and 2, respectively. These routines were written because they were required by the contour shading algorithm implemented in the Fortran90 module. It was a relatively simple matter to write these two functions because they are essentially just wrappers around functions contained in the X11 graphics library. Finally, there are a few routines which are available in the MPE library but currently do not have Fortran bindings. These routines provide functionality for drawing character strings and providing input via the mouse.

2.2 Portability and Language Compatability Issues

In general, the MPE graphics library is portable. Unfortunately for the Fortran programmer, the main sources of dependency on a specific operating system are found in the Fortran

```
subroutine MPE_Fill_polygon(handle, nvertx, x, y, color, mpe_err)
  integer handle          ! handle for MPE X11 window
  integer  nvertx         ! number of vertices in polygon
  integer, dimension(nvertx):: x, y ! screen coordinates of vertices
  integer   color         ! color with which to fill polygon
  integer mpe_err ! MPE error code

subroutine MPE_Fill_triangle(handle, x1, y1, x2, y2, x3, y3, &
           color, mpe_err)
  integer handle          ! handle for MPE X11 window
  integer x1, y1, x2, y2, x3, y3     ! triangle vertices
  integer   color         ! color with which to fill triangle
  integer mpe_err ! MPE error code
```

Figure 1: Fortran bindings for the new MPE graphics routines.

```
int MPE_Fill_polygon(handle, nvertx, x, y, color)
  MPE_XGraph handle;
  int        *x, *y;
  int nvertx;
  MPE_Color  color;

int MPE_Fill_triangle(handle, x1, y1, x2, y2, x3, y3, color)
  MPE_XGraph handle;
  int        x1, y1, x2, y2, x3, y3;
  MPE_Color  color;
```

Figure 2: C bindings for the new MPE graphics routines.

Fortran	C		
	Unicos M/K	AIX	IRIX
`call foo()`	`void FOO()`	`void foo()`	`void foo_()`

Table 2: How to name a C function when called from Fortran.

bindings. For example, Table 2 illustrates how a C function must be named on different platforms when called from Fortran. Note that the C compiler is sensitive to the text case (*i.e.* upper or lower) and sometimes the Fortran compiler appends an underscore to the function name. Another source of dependency is associated with the fact that the Fortran language specifies call-by-reference, whereas C specifies call-by-value when passing arguments. In other words, an argument to a C function is always a value, whereas an argument to a Fortran subroutine is always an address to a value. This means that the arguments to a C function which is called from Fortran must always be a pointer, so that it represents an address that can be passed by Fortran.

A portability issue arises when a pointer is converted to an integer (*e.g.*, for passing the graphics handle back to the Fortran caller, as is done in the `MPE_Open_graphics()` function). Under the 64-bit IRIX ABI, a `long int` is required to hold the address contained in a pointer. On other operating systems, a standard `int` may suffice.

Finally, there is an include file, `mpef.h`, which contains parameter definitions required by the MPE library. At the time of this writing, the file which is included in the official distribution is not compatible with both free-form Fortran90 and old-style, fixed-form Fortran77 source files. The modifications to remedy this are simple, and consist of altering the comment and line continuation syntax. A modified version of this file is included with the module distribution.

The relationship of the MPE graphics library with a finite element application, MPI, and the X11 library is illustrated schematically in Figure 3. At the lowest level, the X11 system library is called by the MPE routines to perform the actual drawing and display. The MPI message-passing library may be needed at more than one level. First, it is needed at the highest level by the finite element application to perform the communication required for the simulation. Next, it may be required by the solution contour calculator in order to determine a global bounding box for the simulation domain, solution extrema, *etc.* (This is somewhat of a philosophical issue. In the module described in this report, this communication is performed by the finite element application and is passed to the contour calculator.) Finally, MPI is not currently required by the MPE graphics library but may be in the future in order to support collective operations [2].

3 A Fortran90 module

The module described in this work, `mpe_gfx.f90` is presented as a simple example of how to use the MPE graphics library. It is hoped that it will be deemed useful to other applications programmers, either in and of itself or as a template for the construction of other modules. Towards this end, the complete module is listed in the Appendix. The problem statement that the module addresses is as follows: given a scalar field defined on an unstructured, two-

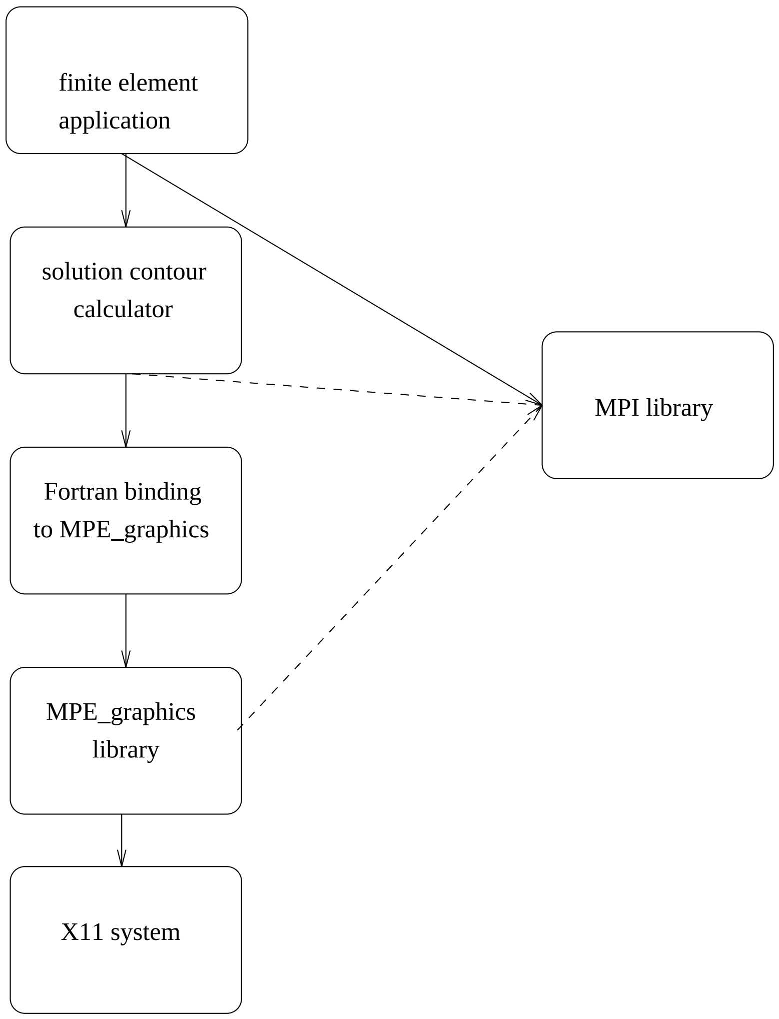
finite element application
solution contour calculator
Fortran binding to MPE_graphics
MPE_graphics library
X11 system
MPI library

Figure 3: Relationship of MPE graphics library with other application components.

dimensional, triangular mesh which is distributed among many processors, draw a colored contour representation of the associated surface. This plot could be drawn at every time step of a distributed-memory, time-dependent calculation in order to observe the time evolution of the solution as it is calculated.

There are two user-defined, integer parameters which must be set at compile time. The first, `rkind`, is the Fortran90 "kind" to be used for real data. For example, set `kind = 8` to obtain 64-bit real variables. The second parameter, `mpe_screen_max`, defines the maximum edge length (in pixels) of the X-window in which the plot will be drawn.

The module has three subroutines which may be called by the user: `init_mpe_gfx()`, `finish_mpe_gfx()`, and `mpe_cnt()`. The bindings for these routines are given in Figure 4.

Subroutine `init_mpe_gfx()` is a wrapper around `MPE_Open_graphics()`, and also performs some preprocessing. The basic algorithm is as follows:

1. obtain the X-window `DISPLAY` environment variable
2. compute size of X-window in which to display plot
3. allocate integer arrays to store the screen coordinates
4. convert from real-valued (x, y) coordinates to screen coordinates
5. call `MPE_Open_graphics()`
6. optionally call `MPE_Capturefile()`

The X-window size in step 2 above is determined from the value of `mpe_screen_max` and the computed aspect ratio of the given domain. The allocated arrays in step 3 are defined in the module header as `private`, so that they cannot be accessed by routines outside of the module. The call to `MPE_Open_graphics()` actually opens the X-window on the display screen.

Next, the main routine is `mpe_cnt()`, which performs the following steps:

1. allocate array to hold color map
2. call `MPE_Make_color_array()`
3. fill X-window with white
4. draw mesh boundaries
5. loop over each triangle
   - (a) loop over the contour values
   - (b) set color for this contour
   - (c) draw contours for this triangle
6. call `MPE_Update()`

```
  subroutine init_mpe_gfx(capture, npoints, comm, myid, x, y, &
             xmin, ymin, xmax, ymax)
    logical capture            ! if true, then capture plots to files
    integer npoints            ! local number of grid points
    integer comm               ! MPI communicator.
    integer myid               ! rank of this process.
    real(kind=rkind), dimension(npoints):: x,y ! real coords of local grid
    real(kind=rkind):: xmin, xmax, ymin, ymax ! bounding box of global grid


  subroutine mpe_cnt( ifill, npoin, ntri, nedge, nvalues, elist, &
             nodes, prop, testval)
    integer ifill              ! if 0 draw colored lines, else
!                                draw color-filled plot
    integer npoin              ! number of points in local grid
    integer ntri               ! number of triangles in local grid
    integer nedge              ! number of boundary edges in local grid
    integer nvalues            ! number of contour values to draw.
    real(kind=rkind)::  prop(npoin) ! array to contour
    integer:: nodes(3,ntri)    ! triangle list
    integer:: elist(2,nedge)   ! boundary edge list
    real(kind=rkind):: testval(nvalues)   ! contour values to test for


  subroutine finish_mpe_gfx
    !note:  no arguments.
```

Figure 4: Fortran90 bindings for user-callable routines in the module, listed in the order in which they should be called.

In step 3 above, the window is painted white in order to clear the plot from a previous time step. The contours may be drawn in step 5c in one of two ways: either as colored contour lines or as color-filled polygons. This action is determined at runtime via an argument of `mpe_cnt()`. The contour algorithm is very simple: for each triangle in the mesh, the property values at the vertices are compared with each of the given contour values. The color is set according to the active contour value, and contours are drawn if the test value lies between two vertex values. A call to `MPE_Update()` after each processor has drawn its portion of the domain ensures that the plot buffer is flushed before the next timestep.

After all plots have been made, a call to `finish_mpe_gfx()` deallocates the screen coordinate arrays and closes the plot window by a call to `MPE_Close_graphics()`.

4 Image Conversions

As stated above, the `MPE_Capturefile()` routine is used to determine if displayed images will be saved to individual files in the `xwd` format. Since it may require several hours of compute time in order to complete a program's execution, having such a record of graphical output would allow a post mortem review of generated data. There are many tools and programs available that are able to view the `xwd` format and these may be sufficient.

However, if you are unable to view `xwd` files, the following subsections give several possible methods of converting these files to a desirable and compatible format. Descriptions and examples of two major tools, namely `dmconvert` and the Image Tools suite, will be given. The former is a standard utility available on SGI workstations and the latter is available from the San Diego Supercomputer Center (SDSC) by anonymous FTP at `ftp://ftp.sdsc.edu/pub/sdsc/graphics/imtools/` in source or pre-compiled binary form. Full details of all the tools discussed below are available within `man` pages.

For sake of example, we shall assume that twenty `xwd` files have been generated. These files are named `flick000.xwd`, `flick001.xwd`, ..., `flick019.xwd`.

4.1 Converting Single Images

4.1.1 `dmconvert`

The `dmconvert` utility on SGI workstations is able to convert between 38 different graphical image formats. There are restrictions on some formats which only allow them to be exclusively used as either input or output. These restrictions are noted on the `man` page for `dmconvert`.

The command to convert the first example `xwd` file into a GIF file would be,

```
dmconvert -f gif -p video flick000.xwd flick000.gif
```

where the `-f gif` denotes the image format type to be converted to, `-p video` denotes the track type to convert (`audio` is the other possibility), and `flick000.xwd` and `flick000.gif` are the input and output files, respectively.

One caveat that must be mentioned: in our experiments with the `dmconvert` utility, all converted files came out in grayscale. That is, while the `xwd` file had all the colors present that were displyed during program execution, after running them through the conversion process, the files were rendered in black, white and shades of gray. We remark that some combination of command line options currently unknown to us may correct this problem.

4.1.2 `imconv`

The `imconv` tool within the SDSC Image Tools suite supports slightly fewer image formats than `dmconvert`. (The `imformats` utility is used to list out all available formats.) However, in our experiments with the suite, all of the target image conversions from the `xwd` format were able to preserve the colors found in the original image.

The command to convert the first example `xwd` file into a GIF file would be,

```
imconv -infile flick000.xwd -outfile flick000.gif
```

where `-infile` and `-outfile` are optional and would only be needed if the input and output file names were embedded within a list of command line arguments.

4.2 Converting Multiple Images

4.2.1 imcat

The `imcat` utility concatenates multiple image files into a single file. Most likely, the format for such a target file will be Tagged Image File Format (`tiff`) or Hierarchical Data File (`hdf`). The command to concatenate all of the example `xwd` files into a TIFF file would be,

```
imcat -frames 0-19 flick%03d.xwd -outfile flicks.tiff
```

where `-frames 0-19 flick%03d.xwd` is an example of the implicit file naming scheme used in utilities of the Image Tools suite. This implicit naming convention is described in more detail below.

4.2.2 imstoryboard

The `imstoryboard` is able to arrange a set of image files into a single image file. Individual input images are placed into a storyboard (or grid) ordering within the output file. This output would be useful to view an entire set of generated images and also be able to track the progression of images through time.

The command to create a GIF storyboard containing the complete set of example `xwd` files would be,

```
imstoryboard -frames 0-19 flick%03d.xwd -outfile flickstory.gif
```

where `-frames 0-19 flick%03d.xwd` is an example of the implicit file naming scheme. There are additional flags and command arguments that control the size, shape and placement of images within the grid framework.

The special character code "%d" is used to define a multiple file name template in some of the Image Tools utilities. This code, much like the C language `printf` output edit descriptor, is replaced by the range of numbers specified in the `-frames` argument. The number of digits used in this replacement is controllable by the user. For example, the `%03d` used above specifies a zero-filled, three-digit output. This corresponds to the file names of the example input image files.

4.2.3 Making Movies

Besides converting from one image format to another, the `dmconvert` tool is able to concatenate multiple images into a an animated presentation. Thus, it would be possible to convert the entire set of example `xwd` image files into either an AVI, MPEG, QuickTime or SGI movie format. However, in our experiments with the `dmconvert` utility, we were unable to create converted files with the colors contained in the original `xwd` files.

We were able to convert color `rgb` files from `imconv` and use `dmconvert` to create a color movie file. The following process would be used to make a MPEG movie of all the example `xwd` files:

```
#/bin/csh -f
foreach file (*.xwd)
   set rootname=$file:r
   imconv $rootname.xwd $rootname.rgb
end
```

Figure 5: `xwd` to `rgb` conversion script

1. Convert all `xwd` files to `rgb` files. Since the `imconv` can only work with a single input and output file at one time, the script file shown in Figure 5 , when executed, will convert all files suffixed with `xwd` into correspondingly named `rgb` files.
2. Convert all `rgb` files to a single MPEG movie file. the command to do this would be,

   ```
   dmconvert -f mpeg1v -p video flick0##.rgb flick.mpg
   ```

   where `flick0##.rgb` is a template for multiple files named with a sequential numbering scheme. The string of "#" characters in the template indicates the maximum size of the field to replaced by integers in sequence, left-padded with zeros to the field length, starting at zero (0) and incremented by one (1). These sequential file names are used in numerical order until no more files match the template. Thus, the template given above would match the files `flick000.rgb` to `flick099.rgb`.

5 Summary

A brief description of how the MPE graphics library may be used to observe the time evolution of a distributed-memory, MPI-based simulation has been presented. The MPE library is small, and is therefore relatively easy to learn. In particular, a Fortran90 module which draws contours of a three-dimensional surface on unstructured triangular meshes has been written as an example. This module, which is listed in the Appendix, is available at `http://www.erc.msstate.edu/~swb/Tools`. A brief overview on image file conversion and animation has also been presented. In particular, we have described how `xwd` image files which were obtained via the `MPE_Capturefile()` function may be animated for later viewing.

A potentially interesting application of this library lies in the area of remote visualization. Because the drawing primitives are based on the ubiquitous X11 system, they can be displayed on practically any hardware. The X11 library has been ported to both Macintosh and Intel-based personal computers, and is included with virtually all Unix operating systems. Furthermore clients can be reasonably displayed over telephone connections with modern, high-speed (*e.g.* 19,200 baud) modems.

It is unfortunate that there is so little documentation on the MPE library. If a user's guide were included with the distribution, or made available at the MPICH homepage (`http://www.mcs.anl.gov/mpi/mpich`), then we believe that the existence and utility of this library would be more widely appreciated.

References

[1] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, London, 1994.

[2] MPE_open_graphics unix man page. MPE Graphics Library Unix Programmer's Manual.

Appendix

```
module mpe_gfx
  implicit none

  include 'mpef.h'

  integer, parameter :: rkind = 8 ! kind of real data
  integer, private, parameter :: mpe_screen_max = 1000 ! max window edge length
! -----------------------------------------------------------------------
! end of user-initialized data
  integer, private :: mpe_comm             ! MPI communicator.
  integer, private :: mpe_myid             ! rank of this process.
  integer, private :: l_winx    ! length of window in x
  integer, private :: l_winy    ! length of window in y

  integer mpe_err              ! error code from mpe package

  real(kind = rkind), private :: gxmin, gxmax, gymin, gymax ! bounding box of
                                ! global grid

  integer (kind = 8), private ::  mpe_handle ! graphics handle for
                                ! MPE gfx routines.
  integer, private, allocatable, dimension(:) :: mpe_xs !screen x-coords
  integer, private, allocatable, dimension(:) :: mpe_ys !screen y-coords

contains

! begin  init_mpe_gfx
  subroutine init_mpe_gfx(capture, npoints, comm, myid, &
       x, y, xmin, ymin, xmax, ymax)
    logical capture            ! if true, then capture plots to files
    integer npoints            ! local number of grid points
    integer comm               ! MPI communicator.
    integer myid               ! rank of this process.
    real(kind = rkind), dimension(npoints) :: x,y ! real coords of local grid
    real(kind = rkind) :: xmin, xmax, ymin, ymax ! bounding box of global grid

    real(kind = rkind) :: xdis, ydis

    character(len=81) :: xdisplay ! DISPLAY environment variable

    gxmin = xmin
    gymin = ymin
    gxmax = xmax
    gymax = ymax
    mpe_comm = comm
    mpe_myid = myid

!   allocate the screen coordinate arrays
```

```
    allocate( mpe_xs(npoints), mpe_ys(npoints) )

!   get the xwindow DISPLAY environment variable
    call getenv("DISPLAY", xdisplay)
    xdisplay = trim(xdisplay)//char(0) ! append the null character for C

! transform from physical coordinates to screen coordinates
    xdis = gxmax - gxmin
    ydis = gymax - gymin

!   figure the X window sizes
    if(xdis > ydis) then
       l_winx = mpe_screen_max
       l_winy = ydis*l_winx/xdis
    else
       l_winy = mpe_screen_max
       l_winx = xdis*l_winy/ydis
    end if

    mpe_xs(1:npoints) = (x(1:npoints) - gxmin)/xdis*l_winx
    mpe_ys(1:npoints) = (y(1:npoints) - gymin)/ydis*l_winy

    l_winx = 1.1*l_winx
    l_winy = 1.1*l_winy

!   open the mpe gfx package

    call mpe_open_graphics(mpe_handle,mpe_comm, xdisplay, &
         -1,-1, l_winx, l_winy, 0, mpe_err)
    if(capture) call MPE_Capturefile(mpe_handle, "flick",1,mpe_err)

  end subroutine init_mpe_gfx

! begin finish_mpe_gfx
  subroutine finish_mpe_gfx

    deallocate(mpe_xs,mpe_ys)
    call mpe_close_graphics(mpe_handle,mpe_err)

  end subroutine finish_mpe_gfx

! begin mpe_cnt
  subroutine mpe_cnt(ifill, npoin, ntri, nedge, nvalues, elist, &
       nodes, prop, testval)

    integer ifill              ! if 0 draw colored lines, else
!                                draw color-filled plot
    integer npoin              ! number of points in local grid
    integer ntri               ! number of triangles in local grid
    integer nedge              ! number of boundary edges in local grid
    integer nvalues             ! number of countour values to draw.
    real(kind=rkind) ::  prop(npoin) ! array to contour
    integer :: nodes(3,ntri)    ! triangle list
    integer :: elist(2,nedge)   ! boundary edge list
    real(kind=rkind) :: testval(nvalues)   ! contour values to test for
!------------------------------------------------------------------------
!     Local Variables:

    real(kind=rkind) ::  aaa(3)
    integer li, lo
    real(kind=rkind) :: xmin, xmax, ymin, ymax, pmin, pmax
    real(kind=rkind) :: xmin1, xmax1, ymin1, ymax1
    real(kind=rkind) :: percent, xdis, ydis
    integer ileft, iright, ibot, itop
    integer icnt, ie
    integer ip1, ip2, ip3
    real(kind=rkind) :: slope
    real(kind=rkind) :: xstr, ystr, dystr
    integer leng
    character*40 string
    integer ichrlen
    integer shade
    integer xtri(3), ytri(3)
    integer, allocatable, dimension(:) :: colours ! color array

    pmin = minval(prop(1:npoin))
    pmax = maxval(prop(1:npoin))

! figure testvalues
    if (nvalues .lt. 0) then
       nvalues = -nvalues
    end if

    allocate(colours(0:nvalues))

! clear the screen

    if(0 == ifill .and. 0 == mpe_myid) then
       call MPE_Fill_rectangle(mpe_handle, 0, 0, l_winx, l_winy, &
            MPE_WHITE, mpe_err)
       call mpe_update(mpe_handle,mpe_err)
    end if

! set the colormap
    call MPE_Make_color_array(mpe_handle, nvalues, colours, mpe_err)

    if(mpe_err /= 0 )then
       write(*,*)'mpe_err: ',mpe_err
    end if
! draw in the boundaries
```

```
    do ie = 1, nedge
       ip1 = elist(1,ie)
       ip2 = elist(2,ie)
!      draw the edge
       call MPE_Draw_line(mpe_handle, mpe_xs(ip1), mpe_ys(ip1), &
            mpe_xs(ip2), mpe_ys(ip2), MPE_BLACK, mpe_err)
    end do

!     loop over the elements

    if(0 == ifill) then
       do  ie = 1, ntri
!        get the index of each node
          ip1 = nodes(1,ie)
          ip2 = nodes(2,ie)
          ip3 = nodes(3,ie)
          xtri(1) = mpe_xs(ip1)
          ytri(1) = mpe_ys(ip1)
          aaa(1) = prop(ip1)
          xtri(2) = mpe_xs(ip2)
          ytri(2) = mpe_ys(ip2)
          aaa(2) = prop(ip2)
          xtri(3) = mpe_xs(ip3)
          ytri(3) = mpe_ys(ip3)
          aaa(3) = prop(ip3)
          call cline(aaa,xtri,ytri,testval,nvalues,colours)
       end do
    else
       do  ie = 1, ntri
!        get the index of each node
          ip1 = nodes(1,ie)
          ip2 = nodes(2,ie)
          ip3 = nodes(3,ie)
          xtri(1) = mpe_xs(ip1)
          ytri(1) = mpe_ys(ip1)
          aaa(1) = prop(ip1)
          xtri(2) = mpe_xs(ip2)
          ytri(2) = mpe_ys(ip2)
          aaa(2) = prop(ip2)
          xtri(3) = mpe_xs(ip3)
          ytri(3) = mpe_ys(ip3)
          aaa(3) = prop(ip3)
          call cfill(aaa,xtri,ytri,testval,nvalues,colours)
       end do
    end if
    call mpe_update(mpe_handle,mpe_err)

  end subroutine mpe_cnt

! begin cline
```

```
  subroutine cline(aaa,xtri,ytri,testval,nval1,colours)

!     This routine finds all the color contours for a triangle.
!     This routine was written by modifying the above QTRIANG.
!     The difference is that it generates standard contour lines instead
!     of colored polygons.

!     This is done by finding the number of intersections at each side
!     then do the following:

!     CASE 1: No intersection -> do nothing
!     CASE 2: One intersection -> do nothing
!     CASE 3: Two intersection or if two nodes have same value ->
!             plot the line
!     CASE 4: Three intersection ->
!             Find the duplicate pt and plot the other two
!     CASE 5: All three nodes are the same value and are approximately
!             the same as a testvalue


!     variables
!        aaa(3) - scalar values of the triangle
!        xtri(3),ytri(3) - the coordinates of the triangle
!        xint(2),yint(2) - intersections along a side
!        val(2) - values found at the intersection-aal 2/25/91:not used anymore
!        testval(nval1) - values to test for intersections
!        inod - start node for a side
!        jnod - end node for at side
!        ival - the number of the contour
!        nval - max number of contours
!        nval1 - input max number of contours
!        nint - the number of intersections found along a side
!        v1,v2 - values at the vertices along a side
!        vmax,vmin - the min and max contour values
!        dv - difference between v1 and v2

    integer colours(*)
    integer nval1
    real(kind=rkind) aaa(3)
    integer xtri(3),ytri(3),xint(3),yint(3)
    real(kind=rkind) testval(nval1),v1,v2,dv
    integer inod,jnod,ival,nint,nval

!   check to see if nval1 is o.k.
    nval = nval1
    if (1 == nval) nval = 2
    do ival = 1,nval

!     check all 3 nodes of triangle for intersections
!
```

```
       nint = 0
       do inod = 1,3
          jnod = mod(inod,3)+1
          v1 = aaa(inod)
          v2 = aaa(jnod)
          if( (abs(v1-v2) <= 1.0e-07) .and. &
              ( abs(v1-testval(ival)) <= 1.e-07) ) then

!         set the proper color

             call MPE_Draw_line(mpe_handle, xtri(inod), ytri(inod), &
                  xtri(jnod), ytri(jnod), colours(ival), mpe_err)
             exit
          end if
          if ((v1-testval(ival))*(testval(ival)-v2) > 0.0) then
             nint = nint + 1
             dv = v2 - v1
             if (abs(dv) > 1.e-08*testval(ival)) then
                xint(nint) = xtri(inod) + (testval(ival)-v1)/dv * &
                     (xtri(jnod) - xtri(inod))
                yint(nint) = ytri(inod) + (testval(ival)-v1)/dv * &
                     (ytri(jnod) - ytri(inod))
             end if
          endif
       end do

!     Now evaluate cases
!     case 1 - no intersections:do nothing
!     case 2 - 1 intersection:do nothing
!     case 3 - 2 intersections: plot the line

       if(nint==2) then
!               set the proper color

          call MPE_Draw_line(mpe_handle, xint(1),yint(1), &
               xint(2),yint(2), colours(ival), mpe_err)
       endif
!        case 4 - three intersections:find duplicate point
!                and plot the other two.

99     continue

       if( nint==3 ) then
          if( xint(1)/=xint(2) .and. yint(1)/=yint(2) ) then

             call MPE_Draw_line(mpe_handle, xint(1),yint(1), &
                  xint(2),yint(2), colours(ival), mpe_err)
          else
```

```
             call MPE_Draw_line(mpe_handle, xint(1),yint(1), &
                  xint(3),yint(3), colours(ival), mpe_err)
          endif
       end if
!        call mpe_update(mpe_handle,mpe_err)
    end do

  end subroutine cline

! begin cfill
  subroutine cfill(aaa,xtri,ytri,testv,nval1,colours)

!     This routine finds all the color contours for a triangle.
!     This is done by finding the number of intersections at each side
!     then do the following:

!     CASE 1: No intersection -> do nothing
!     CASE 2: No intersection but side is in range ->
!        get both vertices from triangle in directed order
!     CASE 3: Two intersection ->
!        decide if in directed order
!     CASE 4: One intersection ->
!        get correct vertex from triangle
!        decide if in directed order

!     Directed order means that the vertices of the polygon for the color
!     contour are in the same direction as the triangle.  This is to avoid
!     getting Z's rather than rectangles.

!     variables
!        aaa(3) - scalar values of the triangle
!        xtri(3),ytri(3) - the coordinates of the triangle
!        xver(6),yver(6) - the coordinates of the polygon for the color contour
!        xint(2),yint(2) - intersections along a side
!        val(2) - values found at the intersection
!        testval(2) - values to test for intersections
!        inod - start node for a side
!        jnod - end node for at side
!        ival - the number of the contour
!        nval - max number of contours
!        nval1 - input max number of contours
!        nint - the number of intersections found along a side
!        v1,v2 - values at the vertices along a side
!        vmax,vmin - the min and max contour values
!        dv - difference between v1 and v2
!        ja - counter for intersection checking
!        nver - the number of vertices for the polygon for the color contour
!        ordered - if the intersection locations are in directed order
!        found - if there are intersection found along a side
```

```
    integer colours(*)
    integer inod,jnod,ival,nint,nval,nval1,nver,ja
    integer istart,iend
    logical ordered,found
    real(kind=rkind) aaa(3)
    integer xtri(3),ytri(3),xint(2),yint(2), xver(6), yver(6)

    real val(2),testval(2),v1,v2,dv
!!aal 10/18/90 added testv dimensioned at nval
    real(kind=rkind) testv(nval1)
!!aal 10/18/90

!     check to see if nval1 is o.k.
    nval = nval1
    if (nval == 1) nval = 2
    do ival = 1,nval-1
       testval(1) = testv(ival)
       testval(2) = testv(ival+1)
!         testval(1) = vmin + float(ival-1)/float(nval-1)*(vmax-vmin)
!         testval(2) = vmin + float(ival)/float(nval-1)*(vmax-vmin)
       nver = 0
       do inod = 1,3
          nint = 0
          jnod = mod(inod,3)+1
          v1 = aaa(inod)
          v2 = aaa(jnod)
          found = .false.
          do ja = 1,2
             if ((v1-testval(ja))*(testval(ja)-v2) > 0.0) then
                found = .true.
                nint = nint + 1
                val(nint) = testval(ja)
                dv = v2 - v1
                if (abs(dv) < 1.e-6*testval(ja)) dv = 1.0
                xint(nint) = xtri(inod) + (testval(ja)-v1)/dv * &
                     (xtri(jnod) - xtri(inod))
                yint(nint) = ytri(inod) + (testval(ja)-v1)/dv * &
                     (ytri(jnod) - ytri(inod))
             endif
          end do

!           Now evaluate cases
!           CASE 1 and 2: No intersections
          if ( nint == 0) then

!              check if start node is in range (CASE 2)
             if ((testval(1)-v1)*(v1-testval(2)) >= 0.0) then

!                 case 2 side in range
                ordered = .true.
                found = .true.
                xint(1) = xtri(inod)
                yint(1) = ytri(inod)
                xint(2) = xtri(jnod)
                yint(2) = ytri(jnod)
             endif

!               CASE 1 -> do nothing

!            CASE 3 - Two intersections so check to see if ordered.
          else if (nint == 2) then
             ordered = abs(v1-val(1)) < abs(v1-val(2))

!            CASE 4 - One intersection
          else if ((testval(1)-v1)*(v1-testval(2)) >= 0.0) then
             ordered = .false.
             xint(2) = xtri(inod)
             yint(2) = ytri(inod)
          else
             ordered = .true.
             xint(2) = xtri(jnod)
             yint(2) = ytri(jnod)
          endif

!            if intersection were found then store vertices in ver
!            check to see if istart is a new vertex.
          if ( found ) then
             istart = 2
             iend = 1
             if ( ordered ) then
                istart = 1
                iend = 2
             endif
             if ( (nver == 0) .or. ((xver(nver) /= xint(istart)) &
                  .or. (yver(nver) /= yint(istart))) ) then
                nver = nver + 1
                xver(nver) = xint(istart)
                yver(nver) = yint(istart)
             endif
             nver = nver + 1
             xver(nver) = xint(iend)
             yver(nver) = yint(iend)
          endif
       end do
       if ( nver > 0) then
          if ((xver(1)==xver(nver)) .and. (yver(1)==yver(nver))) &
               nver = nver - 1

        call mpe_fill_polygon(mpe_handle, nver, xver,yver, &
             colours(ival), mpe_err)
       endif
    end do
  end subroutine cfill
end module mpe_gfx
```

Work funded by the DoD High Performance Computing Modernization Program CEWES Major Shared Resource Center through

Programming Environment and Training (PET)

Supported by Contract Number: DAHC 94-96-C0002
Nichols Research Corporation

Views, opinions, and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Defense position, policy, or decision unless so designated by other official documentation.

Using the MPE Graphics Library with Fortran90

S.W. Bova* Clay P. Breshears†

March 13, 1998

1 Introduction

The MPE graphics library is part of the MPICH package distributed by Argonne National Laboratory. "It consists of functions that are not in MPI; consistent in style with MPI; freely available; and in the long run will work with any MPI implementation" [1]. This graphics library gives the MPI programmer an easy-to-use, minimal set of routines that can asynchronously draw color graphics to an X11 window during the course of a numerical simulation. Unfortunately, there is little scientific visualization software written for MPE, which places a high burden on the applications programmer. This difficulty is compounded by a dearth of documentation on the library. On the other hand, the library is small and has a shallow learning curve. This report is a brief description of how these graphics routines may be called from a Fortran90 program, written from the perspective of a finite element applications programmer. It was written after gaining one week of experience in the use of the MPE graphics library, thereby demonstrating that the library is compact enough to be easily used if the applications programmer is familiar with graphics programming. A goal of this report is to make the package equally useful to those programmers who are not as familiar with graphics programming. Note it is not the intent of this report to provide a complete programmer's guide to the MPE library.

The discussion begins in Section 2 with a description of some of the library functions. Next, Section 3 describes a Fortran90 module which uses these routines to draw colored contour plots for two-dimensional, unstructured grids. Section 4 presents an overview on methods and software which can be used to save the graphics for later viewing. Finally, a summary of this report is presented in Section 5 with instructions on how to obtain the module and the MPE graphics library.

2 MPE Graphics Library

In this section, a brief discussion of a subset of the MPE graphics library is presented. For the details of the bindings, refer to the Unix `man` pages which are included in the MPE distribution. Portability issues are also discussed.

*CEWES MSRC On-site CFD Lead for PET, Mississippi State University
†CEWES MSRC On-site Parallel Tools Lead for PET, Rice University

`MPE_Open_graphics()`	`MPE_Draw_circle()`
`MPE_CaptureFile()`	`MPE_Add_RGB_color()`
`MPE_Close_graphics()`	`MPE_Draw_point()`
`MPE_Update()`	`MPE_Draw_points()`
`MPE_Draw_line()`	`MPE_Draw_logic()`
`MPE_Make_color_array()`	`MPE_Fill_circle()`
`MPE_Line_thickness()`	`MPE_Fill_rectangle()`
`MPE_Num_colors()`	

Table 1: MPE graphics routines that have Fortran bindings.

2.1 MPE Graphics Routines

The routines given in Table 1 have Fortran bindings and are available in the current release of MPE graphics. The first six of these routines in the left-hand column were used in the Fortran90 module described in Section 3. The purpose of most of the routines should be obvious from their names. The routine **`MPE_Open_graphics()`** should always be called first. It initializes the system and opens a window on a specified X11 display. The last routine called should be **`MPE_Close_graphics()`**, which gracefully shuts down the X11 window. A color map may be easily defined with **`MPE_Make_color_array()`**. This routine returns color indices in a predefined spectrum. (Arbitrary colors may be defined using **`MPE_Add_RGB_color()`**, but this is not necessary if the colors returned by **`MPE_Make_color_array()`** are satisfactory.) Lines between two points are drawn with **`MPE_Draw_line()`**. The purpose of **`MPE_Update()`** is perhaps not as obvious. Graphics requests are buffered in order to improve performance. A call to **`MPE_Update()`** flushes the graphics buffer and ensures that objects which are drawn are actually displayed in the X11 window. Finally, it may be desirable to capture the contents of the X11 window in an image file on disk. This action is toggled by calling **`MPE_Capturefile()`** which uses the X11 utility **`xwd`**. One of the required arguments is a frequency parameter; *e.g.* if this parameter is equal to two, then an image will be captured every other time **`MPE_Update()`** is called.

The following two routines, along with their Fortran bindings, were added for the present work and are not contained in the current MPE release: **`MPE_Fill_triangle()`** and **`MPE_Fill_polygon()`**. Since these routines are not part of the MPE library, their Fortran and C bindings are given in Figures 1 and 2, respectively. These routines were written because they were required by the contour shading algorithm implemented in the Fortran90 module. It was a relatively simple matter to write these two functions because they are essentially just wrappers around functions contained in the X11 graphics library. Finally, there are a few routines which are available in the MPE library but currently do not have Fortran bindings. These routines provide functionality for drawing character strings and providing input via the mouse.

2.2 Portability and Language Compatability Issues

In general, the MPE graphics library is portable. Unfortunately for the Fortran programmer, the main sources of dependency on a specific operating system are found in the Fortran

```
subroutine MPE_Fill_polygon(handle, nvertx, x, y, color, mpe_err)
  integer handle          ! handle for MPE X11 window
  integer  nvertx         ! number of vertices in polygon
  integer, dimension(nvertx):: x, y ! screen coordinates of vertices
  integer   color         ! color with which to fill polygon
  integer mpe_err ! MPE error code

subroutine MPE_Fill_triangle(handle, x1, y1, x2, y2, x3, y3, &
           color, mpe_err)
  integer handle          ! handle for MPE X11 window
  integer x1, y1, x2, y2, x3, y3     ! triangle vertices
  integer   color         ! color with which to fill triangle
  integer mpe_err ! MPE error code
```

Figure 1: Fortran bindings for the new MPE graphics routines.

```
int MPE_Fill_polygon(handle, nvertx, x, y, color)
  MPE_XGraph handle;
  int        *x, *y;
  int nvertx;
  MPE_Color  color;

int MPE_Fill_triangle(handle, x1, y1, x2, y2, x3, y3, color)
  MPE_XGraph handle;
  int        x1, y1, x2, y2, x3, y3;
  MPE_Color  color;
```

Figure 2: C bindings for the new MPE graphics routines.

Fortran	C		
	Unicos M/K	AIX	IRIX
`call foo()`	`void FOO()`	`void foo()`	`void foo_()`

Table 2: How to name a C function when called from Fortran.

bindings. For example, Table 2 illustrates how a C function must be named on different platforms when called from Fortran. Note that the C compiler is sensitive to the text case (*i.e.* upper or lower) and sometimes the Fortran compiler appends an underscore to the function name. Another source of dependency is associated with the fact that the Fortran language specifies call-by-reference, whereas C specifies call-by-value when passing arguments. In other words, an argument to a C function is always a value, whereas an argument to a Fortran subroutine is always an address to a value. This means that the arguments to a C function which is called from Fortran must always be a pointer, so that it represents an address that can be passed by Fortran.

A portability issue arises when a pointer is converted to an integer (*e.g.*, for passing the graphics handle back to the Fortran caller, as is done in the `MPE_Open_graphics()` function). Under the 64-bit IRIX ABI, a `long int` is required to hold the address contained in a pointer. On other operating systems, a standard `int` may suffice.

Finally, there is an include file, `mpef.h`, which contains parameter definitions required by the MPE library. At the time of this writing, the file which is included in the official distribution is not compatible with both free-form Fortran90 and old-style, fixed-form Fortran77 source files. The modifications to remedy this are simple, and consist of altering the comment and line continuation syntax. A modified version of this file is included with the module distribution.

The relationship of the MPE graphics library with a finite element application, MPI, and the X11 library is illustrated schematically in Figure 3. At the lowest level, the X11 system library is called by the MPE routines to perform the actual drawing and display. The MPI message-passing library may be needed at more than one level. First, it is needed at the highest level by the finite element application to perform the communication required for the simulation. Next, it may be required by the solution contour calculator in order to determine a global bounding box for the simulation domain, solution extrema, *etc.* (This is somewhat of a philosophical issue. In the module described in this report, this communication is performed by the finite element application and is passed to the contour calculator.) Finally, MPI is not currently required by the MPE graphics library but may be in the future in order to support collective operations [2].

3 A Fortran90 module

The module described in this work, `mpe_gfx.f90` is presented as a simple example of how to use the MPE graphics library. It is hoped that it will be deemed useful to other applications programmers, either in and of itself or as a template for the construction of other modules. Towards this end, the complete module is listed in the Appendix. The problem statement that the module addresses is as follows: given a scalar field defined on an unstructured, two-

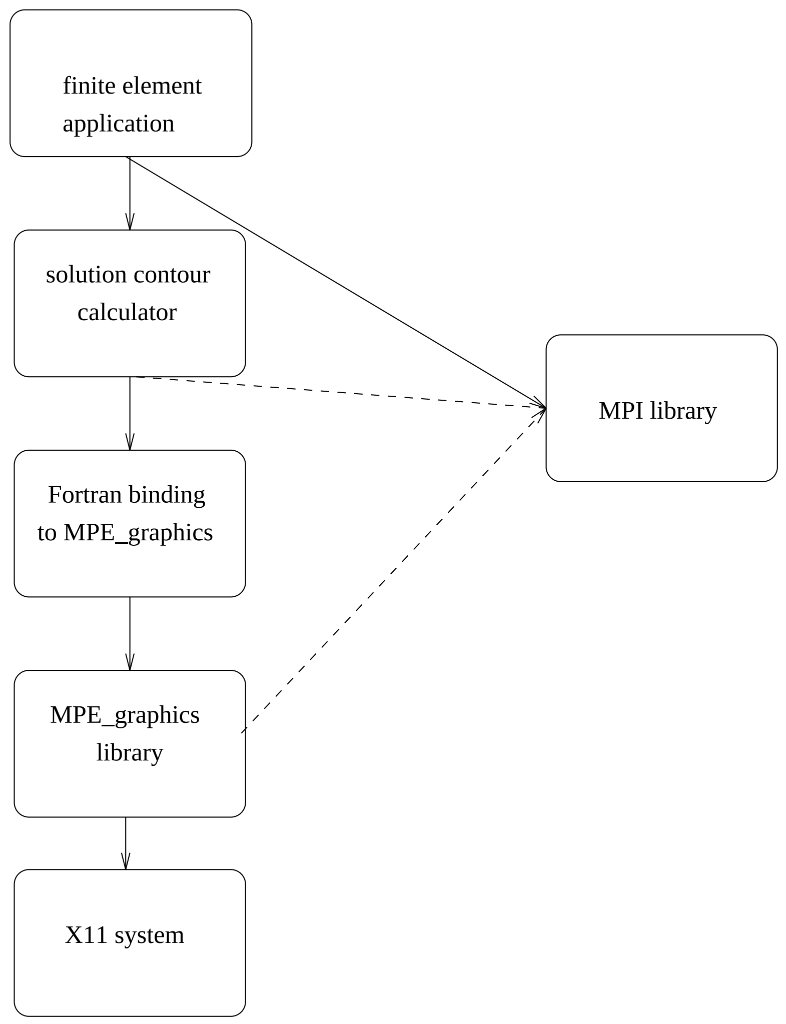
finite element application
solution contour calculator
Fortran binding to MPE_graphics
MPE_graphics library
X11 system
MPI library

Figure 3: Relationship of MPE graphics library with other application components.

dimensional, triangular mesh which is distributed among many processors, draw a colored contour representation of the associated surface. This plot could be drawn at every time step of a distributed-memory, time-dependent calculation in order to observe the time evolution of the solution as it is calculated.

There are two user-defined, integer parameters which must be set at compile time. The first, `rkind`, is the Fortran90 "kind" to be used for real data. For example, set `kind = 8` to obtain 64-bit real variables. The second parameter, `mpe_screen_max`, defines the maximum edge length (in pixels) of the X-window in which the plot will be drawn.

The module has three subroutines which may be called by the user: `init_mpe_gfx()`, `finish_mpe_gfx()`, and `mpe_cnt()`. The bindings for these routines are given in Figure 4.

Subroutine `init_mpe_gfx()` is a wrapper around `MPE_Open_graphics()`, and also performs some preprocessing. The basic algorithm is as follows:

1. obtain the X-window `DISPLAY` environment variable
2. compute size of X-window in which to display plot
3. allocate integer arrays to store the screen coordinates
4. convert from real-valued (x, y) coordinates to screen coordinates
5. call `MPE_Open_graphics()`
6. optionally call `MPE_Capturefile()`

The X-window size in step 2 above is determined from the value of `mpe_screen_max` and the computed aspect ratio of the given domain. The allocated arrays in step 3 are defined in the module header as `private`, so that they cannot be accessed by routines outside of the module. The call to `MPE_Open_graphics()` actually opens the X-window on the display screen.

Next, the main routine is `mpe_cnt()`, which performs the following steps:

1. allocate array to hold color map
2. call `MPE_Make_color_array()`
3. fill X-window with white
4. draw mesh boundaries
5. loop over each triangle
   (a) loop over the contour values
   (b) set color for this contour
   (c) draw contours for this triangle
6. call `MPE_Update()`

```
  subroutine init_mpe_gfx(capture, npoints, comm, myid, x, y, &
            xmin, ymin, xmax, ymax)
    logical capture            ! if true, then capture plots to files
    integer npoints            ! local number of grid points
    integer comm               ! MPI communicator.
    integer myid               ! rank of this process.
    real(kind=rkind), dimension(npoints):: x,y ! real coords of local grid
    real(kind=rkind):: xmin, xmax, ymin, ymax ! bounding box of global grid


  subroutine mpe_cnt( ifill, npoin, ntri, nedge, nvalues, elist, &
            nodes, prop, testval)
    integer ifill              ! if 0 draw colored lines, else
!                                draw color-filled plot
    integer npoin              ! number of points in local grid
    integer ntri               ! number of triangles in local grid
    integer nedge              ! number of boundary edges in local grid
    integer nvalues            ! number of contour values to draw.
    real(kind=rkind)::  prop(npoin) ! array to contour
    integer:: nodes(3,ntri)    ! triangle list
    integer:: elist(2,nedge)   ! boundary edge list
    real(kind=rkind):: testval(nvalues)   ! contour values to test for


  subroutine finish_mpe_gfx
    !note:  no arguments.
```

Figure 4: Fortran90 bindings for user-callable routines in the module, listed in the order in which they should be called.

In step 3 above, the window is painted white in order to clear the plot from a previous time step. The contours may be drawn in step 5c in one of two ways: either as colored contour lines or as color-filled polygons. This action is determined at runtime via an argument of `mpe_cnt()`. The contour algorithm is very simple: for each triangle in the mesh, the property values at the vertices are compared with each of the given contour values. The color is set according to the active contour value, and contours are drawn if the test value lies between two vertex values. A call to `MPE_Update()` after each processor has drawn its portion of the domain ensures that the plot buffer is flushed before the next timestep.

After all plots have been made, a call to `finish_mpe_gfx()` deallocates the screen coordinate arrays and closes the plot window by a call to `MPE_Close_graphics()`.

4 Image Conversions

As stated above, the `MPE_Capturefile()` routine is used to determine if displayed images will be saved to individual files in the `xwd` format. Since it may require several hours of compute time in order to complete a program's execution, having such a record of graphical output would allow a post mortem review of generated data. There are many tools and programs available that are able to view the `xwd` format and these may be sufficient.

However, if you are unable to view `xwd` files, the following subsections give several possible methods of converting these files to a desirable and compatible format. Descriptions and examples of two major tools, namely `dmconvert` and the Image Tools suite, will be given. The former is a standard utility available on SGI workstations and the latter is available from the San Diego Supercomputer Center (SDSC) by anonymous FTP at `ftp://ftp.sdsc.edu/pub/sdsc/graphics/imtools/` in source or pre-compiled binary form. Full details of all the tools discussed below are available within `man` pages.

For sake of example, we shall assume that twenty `xwd` files have been generated. These files are named `flick000.xwd`, `flick001.xwd`, ..., `flick019.xwd`.

4.1 Converting Single Images

4.1.1 `dmconvert`

The `dmconvert` utility on SGI workstations is able to convert between 38 different graphical image formats. There are restrictions on some formats which only allow them to be exclusively used as either input or output. These restrictions are noted on the `man` page for `dmconvert`.

The command to convert the first example `xwd` file into a GIF file would be,

```
dmconvert -f gif -p video flick000.xwd flick000.gif
```

where the `-f gif` denotes the image format type to be converted to, `-p video` denotes the track type to convert (`audio` is the other possibility), and `flick000.xwd` and `flick000.gif` are the input and output files, respectively.

One caveat that must be mentioned: in our experiments with the `dmconvert` utility, all converted files came out in grayscale. That is, while the `xwd` file had all the colors present that were displyed during program execution, after running them through the conversion process, the files were rendered in black, white and shades of gray. We remark that some combination of command line options currently unknown to us may correct this problem.

4.1.2 `imconv`

The `imconv` tool within the SDSC Image Tools suite supports slightly fewer image formats than `dmconvert`. (The `imformats` utility is used to list out all available formats.) However, in our experiments with the suite, all of the target image conversions from the `xwd` format were able to preserve the colors found in the original image.

The command to convert the first example `xwd` file into a GIF file would be,

```
imconv -infile flick000.xwd -outfile flick000.gif
```

where `-infile` and `-outfile` are optional and would only be needed if the input and output file names were embedded within a list of command line arguments.

4.2 Converting Multiple Images

4.2.1 imcat

The `imcat` utility concatenates multiple image files into a single file. Most likely, the format for such a target file will be Tagged Image File Format (`tiff`) or Hierarchical Data File (`hdf`). The command to concatenate all of the example `xwd` files into a TIFF file would be,

```
imcat -frames 0-19 flick%03d.xwd -outfile flicks.tiff
```

where `-frames 0-19 flick%03d.xwd` is an example of the implicit file naming scheme used in utilities of the Image Tools suite. This implicit naming convention is described in more detail below.

4.2.2 imstoryboard

The `imstoryboard` is able to arrange a set of image files into a single image file. Individual input images are placed into a storyboard (or grid) ordering within the output file. This output would be useful to view an entire set of generated images and also be able to track the progression of images through time.

The command to create a GIF storyboard containing the complete set of example `xwd` files would be,

```
imstoryboard -frames 0-19 flick%03d.xwd -outfile flickstory.gif
```

where `-frames 0-19 flick%03d.xwd` is an example of the implicit file naming scheme. There are additional flags and command arguments that control the size, shape and placement of images within the grid framework.

The special character code "%d" is used to define a multiple file name template in some of the Image Tools utilities. This code, much like the C language `printf` output edit descriptor, is replaced by the range of numbers specified in the `-frames` argument. The number of digits used in this replacement is controllable by the user. For example, the `%03d` used above specifies a zero-filled, three-digit output. This corresponds to the file names of the example input image files.

4.2.3 Making Movies

Besides converting from one image format to another, the `dmconvert` tool is able to concatenate multiple images into a an animated presentation. Thus, it would be possible to convert the entire set of example `xwd` image files into either an AVI, MPEG, QuickTime or SGI movie format. However, in our experiments with the `dmconvert` utility, we were unable to create converted files with the colors contained in the original `xwd` files.

We were able to convert color `rgb` files from `imconv` and use `dmconvert` to create a color movie file. The following process would be used to make a MPEG movie of all the example `xwd` files:

```
#/bin/csh -f
foreach file (*.xwd)
   set rootname=$file:r
   imconv $rootname.xwd $rootname.rgb
end
```

Figure 5: `xwd` to `rgb` conversion script

1. Convert all `xwd` files to `rgb` files. Since the `imconv` can only work with a single input and output file at one time, the script file shown in Figure 5 , when executed, will convert all files suffixed with `xwd` into correspondingly named `rgb` files.
2. Convert all `rgb` files to a single MPEG movie file. the command to do this would be,

   ```
   dmconvert -f mpeg1v -p video flick0##.rgb flick.mpg
   ```

   where `flick0##.rgb` is a template for multiple files named with a sequential numbering scheme. The string of "#" characters in the template indicates the maximum size of the field to replaced by integers in sequence, left-padded with zeros to the field length, starting at zero (0) and incremented by one (1). These sequential file names are used in numerical order until no more files match the template. Thus, the template given above would match the files `flick000.rgb` to `flick099.rgb`.

5 Summary

A brief description of how the MPE graphics library may be used to observe the time evolution of a distributed-memory, MPI-based simulation has been presented. The MPE library is small, and is therefore relatively easy to learn. In particular, a Fortran90 module which draws contours of a three-dimensional surface on unstructured triangular meshes has been written as an example. This module, which is listed in the Appendix, is available at `http://www.erc.msstate.edu/~swb/Tools`. A brief overview on image file conversion and animation has also been presented. In particular, we have described how `xwd` image files which were obtained via the `MPE_Capturefile()` function may be animated for later viewing.

A potentially interesting application of this library lies in the area of remote visualization. Because the drawing primitives are based on the ubiquitous X11 system, they can be displayed on practically any hardware. The X11 library has been ported to both Macintosh and Intel-based personal computers, and is included with virtually all Unix operating systems. Furthermore clients can be reasonably displayed over telephone connections with modern, high-speed (*e.g.* 19,200 baud) modems.

It is unfortunate that there is so little documentation on the MPE library. If a user's guide were included with the distribution, or made available at the MPICH homepage (`http://www.mcs.anl.gov/mpi/mpich`), then we believe that the existence and utility of this library would be more widely appreciated.

References

[1] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface.* MIT Press, London, 1994.

[2] MPE_open_graphics unix man page. MPE Graphics Library Unix Programmer's Manual.

Appendix

```
module mpe_gfx
  implicit none

  include 'mpef.h'

  integer, parameter :: rkind = 8 ! kind of real data
  integer, private, parameter :: mpe_screen_max = 1000 ! max window edge length
! ----------------------------------------------------------------------
! end of user-initialized data
  integer, private :: mpe_comm             ! MPI communicator.
  integer, private :: mpe_myid             ! rank of this process.
  integer, private :: l_winx    ! length of window in x
  integer, private :: l_winy    ! length of window in y

  integer mpe_err              ! error code from mpe package

  real(kind = rkind), private :: gxmin, gxmax, gymin, gymax ! bounding box of
                                ! global grid

  integer (kind = 8), private ::  mpe_handle ! graphics handle for
                                ! MPE gfx routines.
  integer, private, allocatable, dimension(:) :: mpe_xs !screen x-coords
  integer, private, allocatable, dimension(:) :: mpe_ys !screen y-coords

contains

! begin  init_mpe_gfx
  subroutine init_mpe_gfx(capture, npoints, comm, myid, &
       x, y, xmin, ymin, xmax, ymax)
    logical capture            ! if true, then capture plots to files
    integer npoints            ! local number of grid points
    integer comm               ! MPI communicator.
    integer myid               ! rank of this process.
    real(kind = rkind), dimension(npoints) :: x,y ! real coords of local grid
    real(kind = rkind) :: xmin, xmax, ymin, ymax ! bounding box of global grid

    real(kind = rkind) :: xdis, ydis

    character(len=81) :: xdisplay ! DISPLAY environment variable

    gxmin = xmin
    gymin = ymin
    gxmax = xmax
    gymax = ymax
    mpe_comm = comm
    mpe_myid = myid

!   allocate the screen coordinate arrays
```

```
    allocate( mpe_xs(npoints), mpe_ys(npoints) )

!   get the xwindow DISPLAY environment variable
    call getenv("DISPLAY", xdisplay)
    xdisplay = trim(xdisplay)//char(0) ! append the null character for C

! transform from physical coordinates to screen coordinates
    xdis = gxmax - gxmin
    ydis = gymax - gymin

!   figure the X window sizes
    if(xdis > ydis) then
       l_winx = mpe_screen_max
       l_winy = ydis*l_winx/xdis
    else
       l_winy = mpe_screen_max
       l_winx = xdis*l_winy/ydis
    end if

    mpe_xs(1:npoints) = (x(1:npoints) - gxmin)/xdis*l_winx
    mpe_ys(1:npoints) = (y(1:npoints) - gymin)/ydis*l_winy

    l_winx = 1.1*l_winx
    l_winy = 1.1*l_winy

!   open the mpe gfx package

    call mpe_open_graphics(mpe_handle,mpe_comm, xdisplay, &
         -1,-1, l_winx, l_winy, 0, mpe_err)
    if(capture) call MPE_Capturefile(mpe_handle, "flick",1,mpe_err)

  end subroutine init_mpe_gfx

! begin finish_mpe_gfx
  subroutine finish_mpe_gfx

    deallocate(mpe_xs,mpe_ys)
    call mpe_close_graphics(mpe_handle,mpe_err)

  end subroutine finish_mpe_gfx

! begin mpe_cnt
  subroutine mpe_cnt(ifill, npoin, ntri, nedge, nvalues, elist, &
       nodes, prop, testval)

    integer ifill              ! if 0 draw colored lines, else
!                                draw color-filled plot
    integer npoin              ! number of points in local grid
    integer ntri               ! number of triangles in local grid
    integer nedge              ! number of boundary edges in local grid
    integer nvalues             ! number of countour values to draw.
    real(kind=rkind) ::  prop(npoin) ! array to contour
    integer :: nodes(3,ntri)    ! triangle list
    integer :: elist(2,nedge)   ! boundary edge list
    real(kind=rkind) :: testval(nvalues)   ! contour values to test for
!-----------------------------------------------------------------------
!     Local Variables:

    real(kind=rkind) ::  aaa(3)
    integer li, lo
    real(kind=rkind) :: xmin, xmax, ymin, ymax, pmin, pmax
    real(kind=rkind) :: xmin1, xmax1, ymin1, ymax1
    real(kind=rkind) :: percent, xdis, ydis
    integer ileft, iright, ibot, itop
    integer icnt, ie
    integer ip1, ip2, ip3
    real(kind=rkind) :: slope
    real(kind=rkind) :: xstr, ystr, dystr
    integer leng
    character*40 string
    integer ichrlen
    integer shade
    integer xtri(3), ytri(3)
    integer, allocatable, dimension(:) :: colours ! color array

    pmin = minval(prop(1:npoin))
    pmax = maxval(prop(1:npoin))

! figure testvalues
    if (nvalues .lt. 0) then
       nvalues = -nvalues
    end if

    allocate(colours(0:nvalues))

! clear the screen

    if(0 == ifill .and. 0 == mpe_myid) then
       call MPE_Fill_rectangle(mpe_handle, 0, 0, l_winx, l_winy, &
            MPE_WHITE, mpe_err)
       call mpe_update(mpe_handle,mpe_err)
    end if

! set the colormap
    call MPE_Make_color_array(mpe_handle, nvalues, colours, mpe_err)

    if(mpe_err /= 0 )then
       write(*,*)'mpe_err: ',mpe_err
    end if
! draw in the boundaries
```

```
    do ie = 1, nedge
       ip1 = elist(1,ie)
       ip2 = elist(2,ie)
!      draw the edge
       call MPE_Draw_line(mpe_handle, mpe_xs(ip1), mpe_ys(ip1), &
            mpe_xs(ip2), mpe_ys(ip2), MPE_BLACK, mpe_err)
    end do

!     loop over the elements

    if(0 == ifill) then
       do  ie = 1, ntri
!        get the index of each node
          ip1 = nodes(1,ie)
          ip2 = nodes(2,ie)
          ip3 = nodes(3,ie)
          xtri(1) = mpe_xs(ip1)
          ytri(1) = mpe_ys(ip1)
          aaa(1) = prop(ip1)
          xtri(2) = mpe_xs(ip2)
          ytri(2) = mpe_ys(ip2)
          aaa(2) = prop(ip2)
          xtri(3) = mpe_xs(ip3)
          ytri(3) = mpe_ys(ip3)
          aaa(3) = prop(ip3)
          call cline(aaa,xtri,ytri,testval,nvalues,colours)
       end do
    else
       do  ie = 1, ntri
!        get the index of each node
          ip1 = nodes(1,ie)
          ip2 = nodes(2,ie)
          ip3 = nodes(3,ie)
          xtri(1) = mpe_xs(ip1)
          ytri(1) = mpe_ys(ip1)
          aaa(1) = prop(ip1)
          xtri(2) = mpe_xs(ip2)
          ytri(2) = mpe_ys(ip2)
          aaa(2) = prop(ip2)
          xtri(3) = mpe_xs(ip3)
          ytri(3) = mpe_ys(ip3)
          aaa(3) = prop(ip3)
          call cfill(aaa,xtri,ytri,testval,nvalues,colours)
       end do
    end if
    call mpe_update(mpe_handle,mpe_err)

  end subroutine mpe_cnt

! begin cline
```

```
  subroutine cline(aaa,xtri,ytri,testval,nval1,colours)

!     This routine finds all the color contours for a triangle.
!     This routine was written by modifying the above QTRIANG.
!     The difference is that it generates standard contour lines instead
!     of colored polygons.

!     This is done by finding the number of intersections at each side
!     then do the following:

!     CASE 1: No intersection -> do nothing
!     CASE 2: One intersection -> do nothing
!     CASE 3: Two intersection or if two nodes have same value ->
!             plot the line
!     CASE 4: Three intersection ->
!             Find the duplicate pt and plot the other two
!     CASE 5: All three nodes are the same value and are approximately
!             the same as a testvalue


!     variables
!        aaa(3) - scalar values of the triangle
!        xtri(3),ytri(3) - the coordinates of the triangle
!        xint(2),yint(2) - intersections along a side
!        val(2) - values found at the intersection-aal 2/25/91:not used anymore
!        testval(nval1) - values to test for intersections
!        inod - start node for a side
!        jnod - end node for at side
!        ival - the number of the contour
!        nval - max number of contours
!        nval1 - input max number of contours
!        nint - the number of intersections found along a side
!        v1,v2 - values at the vertices along a side
!        vmax,vmin - the min and max contour values
!        dv - difference between v1 and v2

    integer colours(*)
    integer nval1
    real(kind=rkind) aaa(3)
    integer xtri(3),ytri(3),xint(3),yint(3)
    real(kind=rkind) testval(nval1),v1,v2,dv
    integer inod,jnod,ival,nint,nval

!   check to see if nval1 is o.k.
    nval = nval1
    if (1 == nval) nval = 2
    do ival = 1,nval

!     check all 3 nodes of triangle for intersections
!
```

```
       nint = 0
       do inod = 1,3
          jnod = mod(inod,3)+1
          v1 = aaa(inod)
          v2 = aaa(jnod)
          if( (abs(v1-v2) <= 1.0e-07) .and. &
               ( abs(v1-testval(ival)) <= 1.e-07) ) then

!         set the proper color

             call MPE_Draw_line(mpe_handle, xtri(inod), ytri(inod), &
                  xtri(jnod), ytri(jnod), colours(ival), mpe_err)
             exit
          end if
          if ((v1-testval(ival))*(testval(ival)-v2) > 0.0) then
             nint = nint + 1
             dv = v2 - v1
             if (abs(dv) > 1.e-08*testval(ival)) then
                xint(nint) = xtri(inod) + (testval(ival)-v1)/dv * &
                     (xtri(jnod) - xtri(inod))
                yint(nint) = ytri(inod) + (testval(ival)-v1)/dv * &
                     (ytri(jnod) - ytri(inod))
             end if
          endif
       end do

!     Now evaluate cases
!     case 1 - no intersections:do nothing
!     case 2 - 1 intersection:do nothing
!     case 3 - 2 intersections: plot the line

       if(nint==2) then
!               set the proper color

          call MPE_Draw_line(mpe_handle, xint(1),yint(1), &
               xint(2),yint(2), colours(ival), mpe_err)
       endif
!        case 4 - three intersections:find duplicate point
!                and plot the other two.

99     continue

       if( nint==3 ) then
          if( xint(1)/=xint(2) .and. yint(1)/=yint(2) ) then

             call MPE_Draw_line(mpe_handle, xint(1),yint(1), &
                  xint(2),yint(2), colours(ival), mpe_err)
          else
```

```
           call MPE_Draw_line(mpe_handle, xint(1),yint(1), &
                xint(3),yint(3), colours(ival), mpe_err)
         endif
      end if
!      call mpe_update(mpe_handle,mpe_err)
   end do

  end subroutine cline

! begin cfill
  subroutine cfill(aaa,xtri,ytri,testv,nval1,colours)

!     This routine finds all the color contours for a triangle.
!     This is done by finding the number of intersections at each side
!     then do the following:

!     CASE 1: No intersection -> do nothing
!     CASE 2: No intersection but side is in range ->
!        get both vertices from triangle in directed order
!     CASE 3: Two intersection ->
!        decide if in directed order
!     CASE 4: One intersection ->
!        get correct vertex from triangle
!        decide if in directed order

!     Directed order means that the vertices of the polygon for the color
!     contour are in the same direction as the triangle.  This is to avoid
!     getting Z's rather than rectangles.

!     variables
!        aaa(3) - scalar values of the triangle
!        xtri(3),ytri(3) - the coordinates of the triangle
!        xver(6),yver(6) - the coordinates of the polygon for the color contour
!        xint(2),yint(2) - intersections along a side
!        val(2) - values found at the intersection
!        testval(2) - values to test for intersections
!        inod - start node for a side
!        jnod - end node for at side
!        ival - the number of the contour
!        nval - max number of contours
!        nval1 - input max number of contours
!        nint - the number of intersections found along a side
!        v1,v2 - values at the vertices along a side
!        vmax,vmin - the min and max contour values
!        dv - difference between v1 and v2
!        ja - counter for intersection checking
!        nver - the number of vertices for the polygon for the color contour
!        ordered - if the intersection locations are in directed order
!        found - if there are intersection found along a side
```

```
    integer colours(*)
    integer inod,jnod,ival,nint,nval,nval1,nver,ja
    integer istart,iend
    logical ordered,found
    real(kind=rkind) aaa(3)
    integer xtri(3),ytri(3),xint(2),yint(2), xver(6), yver(6)

    real val(2),testval(2),v1,v2,dv
!!aal 10/18/90 added testv dimensioned at nval
    real(kind=rkind) testv(nval1)
!!aal 10/18/90

!     check to see if nval1 is o.k.
    nval = nval1
    if (nval == 1) nval = 2
    do ival = 1,nval-1
       testval(1) = testv(ival)
       testval(2) = testv(ival+1)
!         testval(1) = vmin + float(ival-1)/float(nval-1)*(vmax-vmin)
!         testval(2) = vmin + float(ival)/float(nval-1)*(vmax-vmin)
       nver = 0
       do inod = 1,3
          nint = 0
          jnod = mod(inod,3)+1
          v1 = aaa(inod)
          v2 = aaa(jnod)
          found = .false.
          do ja = 1,2
             if ((v1-testval(ja))*(testval(ja)-v2) > 0.0) then
                found = .true.
                nint = nint + 1
                val(nint) = testval(ja)
                dv = v2 - v1
                if (abs(dv) < 1.e-6*testval(ja)) dv = 1.0
                xint(nint) = xtri(inod) + (testval(ja)-v1)/dv * &
                     (xtri(jnod) - xtri(inod))
                yint(nint) = ytri(inod) + (testval(ja)-v1)/dv * &
                     (ytri(jnod) - ytri(inod))
             endif
          end do

!           Now evaluate cases
!           CASE 1 and 2: No intersections
          if ( nint == 0) then

!              check if start node is in range (CASE 2)
             if ((testval(1)-v1)*(v1-testval(2)) >= 0.0) then

!                 case 2 side in range
                ordered = .true.
```

```
               found = .true.
               xint(1) = xtri(inod)
               yint(1) = ytri(inod)
               xint(2) = xtri(jnod)
               yint(2) = ytri(jnod)
            endif

!             CASE 1 -> do nothing

!          CASE 3 - Two intersections so check to see if ordered.
         else if (nint == 2) then
            ordered = abs(v1-val(1)) < abs(v1-val(2))

!          CASE 4 - One intersection
         else if ((testval(1)-v1)*(v1-testval(2)) >= 0.0) then
            ordered = .false.
            xint(2) = xtri(inod)
            yint(2) = ytri(inod)
         else
            ordered = .true.
            xint(2) = xtri(jnod)
            yint(2) = ytri(jnod)
         endif

!          if intersection were found then store vertices in ver
!          check to see if istart is a new vertex.
         if ( found ) then
            istart = 2
            iend = 1
            if ( ordered ) then
               istart = 1
               iend = 2
            endif
            if ( (nver == 0) .or. ((xver(nver) /= xint(istart)) &
                 .or. (yver(nver) /= yint(istart))) ) then
               nver = nver + 1
               xver(nver) = xint(istart)
               yver(nver) = yint(istart)
            endif
            nver = nver + 1
            xver(nver) = xint(iend)
            yver(nver) = yint(iend)
         endif
      end do
      if ( nver > 0) then
         if ((xver(1)==xver(nver)) .and. (yver(1)==yver(nver))) &
              nver = nver - 1

       call mpe_fill_polygon(mpe_handle, nver, xver,yver, &
            colours(ival), mpe_err)
       endif
    end do
  end subroutine cfill
end module mpe_gfx
```